IMDB Replication

In 4.4

In 4.4 all data required by services was replicated in IMDB (http://sourceforge.net/projects/fastdb/) and files. A change in some entities triggers a replication of relevant data. IMDB data was first written to XML files and then copied over by supervisor to specific locations. The same mechanism was in place for files. Some problem arose when dealing with a large set of users. User retrieval from the PGSQL database was not optimized and XML creation was limited by available memory and transfer via XML-RPC timed out. Prior to 4.4 the number of users supported was in the hundreds. Trying to replicate a large number of users took literally hours, in the happy case it worked at all. A lot of effort was made in 4.4 in order to optimize the SQL queries and XML-RPC transfer. 4.4 supports now thousands of users, with reasonable replication time. The replication mechanism is explained more in depth in SipXecs replication mechanism.

4.6: Mongo

In 4.6 the replication concept remained the same: sipXconfig gathered data from the UI, stores it in the SIPXCONFIG PostgreSQL database and writes it to a location accessible by other services. However, IMDB replication is centered around “reblicable entities”. Still there are configuration files but they are handled now by **cfengine**. Those files are small and do not create OOM on their creation or XML-RPC problems on their transfer. For instance, dialing rules and service configurations are still handled this way.
However, a decision was taken to change the IMDB from **fastDB** to **MongoDB**. MongoDB has the advantage of being created to handle large amount of data (mongo comes form humongous) and having its own replication mechanism across nodes. We are also taking advantage of the Mongo's *replica set* concept. You can read more on MongoDB on their website: http://www.mongodb.org.
One main advantages of the change was that unlike versions prior to 4.6 when editing a user triggered a replication of all users (XMLs held user data in bulk), in 4.6 changing a replicable entity will trigger the insertion in Mongo of just that entity. Replication time for this kind of operation was reduced to a few milliseconds. Regenerating the whole Mongo *entity* DB takes more time but the time is measured in minutes for even over 30.000 users on a machine with reasonable resources.

Replicable entities

A **replicable** entity is an entity that will be written to **MongoDB** in its own document that can be retrieved by any service at any location. Such an entity (read Java object) will basically implement the `org.sipfoundry.sipxconfig.common.Replicable` interface. For instance, `org.sipfoundry.sipxconfig.common.User` is a replicable entity. We take for granted that each *Replicable* is a *BeanWithId*. *Replicable* interface has a few methods that an entity must implement.

```
/**
 * Set of {@link DataSet}s to be considered for this entity
 * @return
 */
public Set<DataSet> getDataSets();
/**
 * Identity of the Mongo entity. It will go in the "ident" field.
 * Not all entities require it.
 * @param domainName
 * @return
 */
public String getIdentity(String domainName);
/**
 * Returns a collection of aliases to go in the "als" field.
 * @param domainName
 * @return
 */
public Collection<AliasMapping> getAliasMappings(String domainName);

/**
 * Return true if this entity is to be considered by {link ValidUsers.getValidUsers()}
 * (required by IVR)
 * @return
 */
public boolean isValidUser();
/**
 * Returns a Map of properties to be inserted in Mongo as is.
 * Key is the name of the field, the value is the object to be inserted.
 * @param domain
 * @return
 */
public Map<String, Object> getMongoProperties(String domain);
```

A replicable entity may or may mot define a set of *DataSet_s. A _DataSet* may be regarded as a set of common properties that are written to the Mongo document through a *DataSet generator*. Each *DataSet* has its own DataSet generator that extends `org.sipfoundry.sipxconfig.commserver.imdb.AbstractDataSetGenerator`. *AbstractDataSetGenerator* is used only to inject some common properties (like the *CoreContext*, the *SIP domain*) and to define abstract methods that dataset generators need to implement.
Replicable entities are required to implement the methods however, not for all of them methods are relevant. For instance, not all replicables require an identity or not all replicables have aliases. Maybe this interface should be refactored.

DataSet Generators

The DataSet generators are in *org.sipfoundry.sipxconfig.commserver.imdb* package and extend `org.sipfoundry.sipxconfig.commserver.imdb.AbstractDataSetGenerator`. They are responsible for actually preparing the document that will be written to Mongo. The actual write will be done in *ReplicationManagerImpl*.
For instance, `org.sipfoundry.sipxconfig.commserver.imdb.Aliases` generate method will retrieve all aliases of the entity and add them to the object in a well defined structure. `org.sipfoundry.sipxconfig.admin.commserver.imdb.Mailstore.generate(Replicable entity, DBObject top)` retrieves all information pertinent to a user's mailstore like email address, IMAP server configuration, etc. All the information written to Mongo in a document was once stored in files. For a user we had information scattered around in different files, now the most part is kept in Mongo.
If you take a look at `org.sipfoundry.sipxconfig.commserver.imdb.SpeedDials` you will see good examples of constructing the Mongo document object. You can also check out Mongo Java API (http://api.mongodb.org/java/2.6.3/).

MongoConstants

`org.sipfoundry.commons.mongo.MongoConstants` is the common place where field names are defined. *sipXcommons* project is accessible by any java project.

ReplicationTrigger

Until Mongo introduction replication triggers were scattered across different implementations. While not perfect, we tried to define a common place for all replication triggers, at least for Mongo replications. As in *sipXconfig*, save*(Object) and delete*(Object) methods are intercepted we figured this would be a good way to trigger Mongo replications. This is where `org.sipfoundry.sipxconfig.admin.commserver.imdb.ReplicationTrigger` comes into play. It implements *DaoEventListener* interface which defines two methods - *public void onSave(Object entity)* and `public void onDelete(Object entity)` that get triggered by saving or deleting an Object. *ReplicationTrigger* implementation of the 2 methods will mainly call the replication manager based on some conditions.
However, as it is always the case with helper classes this one got clogged with code that had nothing to do with replication in general or with replication code that could easily stay within a manager. So we decided to keep `ReplicationTrigger` class just for triggering replication of Replicable entities and for entitites that hold a bunch of users such as `Group` and `Branch`. I kept these together since we treat replication of users separately for performance reasons, as explained above. So, I'd say that ReplicationTrigger is pretty much a closed class, i wouldn't put code in there unless necessary. Actually, I can't think of any code that would fit in there except if we invent a new entity that would hold many users.

Replication Manager

The replication manager bean (`org.sipfoundry.sipxconfig.admin.commserver.imdb.ReplicationManagerImpl`) main function is to initiate the *DBCollection*, initiate the Mongo document object, delegate the construction of the Mongo document to the relevant DataSet generator and finally save the document to Mongo. It is also responsible for the parallel asynchronous replication of groups of replicable entities (groups, branches) and for the regeneration of the entire entity collection. It used to also hold the business methods to actually build the service config files that needed to be replicated and delegated the actual replication to the supervisor on the specified location. In 4.6 files are replicated using newly introduced *cfengine* and is not object of this page.

`public static String getEntityId(Replicable entity)` method defines the unique identity of the document as defined here: http://www.mongodb.org/display/DOCS/Object+IDs. The identity is unique and it is formed by the simple class name of the entity and the id of the object from PostgreSQL. It is by default indexed by Mongo.

`public DBObject findOrCreate(Replicable entity)` method is a business method that finds an object in the Mongo entity collection or creates it if it does not exist. It also adds some properties to the object like *MongoConstants.IDENTITY* (ident) or properties defined by each replicable entity in the *getMongoProperties* method.

`public DBCollection getDbCollection()` will instantiate the Mongo collection in which the entities are written (currently **imdb.entity**); also here we can apply some properties to the mongo connection/collection. For instance, we may define indexes:

```
DBObject indexes = new BasicDBObject();
          indexes.put(MongoConstants.TIMESTAMP, 1);
          entity.createIndex(indexes);
```

Replicate All - regenerate Mongo DB

Although great care has been taken in order to keep Postgres and Mongo in sync, some effort was put into correctly regenerating the entire configuration data if the admin feels that some of it is incorrect. The admin may access the locations page, select the desired location and trigger a "*Send profile*". Send profiles will regenerate all configurations for the selected location. If and only if the admin chooses to send profiles to the primary location, then a full regeneration of the Mongo *entity* database will take place. *ReplicationManager*'s `public void replicateAllData()` is the method responsible for this operation. It has 2 parts: first rewrite all the users, then rewrite all other replicable entities (OpenACD objects, conferences, authorization codes, etc). The separation was done because the most problems are put by the large number of users. We want to accommodate systems with tens of thousands of users, the other entities reach only a fraction of that number. Except users, all other entities are retrieved by providers which will search them and return them form the Postgres DB. The providers, are beans that implement *ReplicableProvider* interface which only method is `List<Replicable> getReplicables()`. Each provider is responsible to implement the method and gather all replicable entities and to return them to the ReplicationManager which in turn will write them to Mongo using the mechanism described above.

Asynchronous Parallel Processing

As mentioned before we may deal with groups containing a large set of replicable entities. Be it that we have a large group of users, or a branch containing a lot of users, or simply we want to regenerate all user data. As the reduction of the replication time was one of the goals (keeping the server functioning well being another) why not take advantage of the multi core processors that nowadays are omnipresent even in personal computers, not to mention servers. `private synchronized void doParallelAsyncReplication(int membersCount, Class<? extends ReplicationWorker> cls, Object type)` is the method responsible with the parallel and asynchronous replication of such groups. It is synchronized because we don't want multiple threads doing heavy replication stuff at the same time (let's say we change a group with 20.000 users a few times in a row - note that changing a group will trigger a replication of all users in the group). It is also aynchronous due to the fact for heavy replication stuff, for instance when we hit "send profiles" we want the control taken to the page instantaneously and not receiving a timeout.
The admin can control the number of threads the replication to use and the number of users each thread will retrieve. All this is configurable from a properties file: **{INSTALL}/etc/sipxpbx/sipxconfig.properties**. The properties are (and the values are the default):

```
#mongo replication multi-threading
replicationManagerImpl.nThreads=2
replicationManagerImpl.pageSize=1000
replicationManagerImpl.useDynamicPageSize=false
```

Using Spring these values are injected into *ReplicationManagerImpl*.

- nThreads - number of threads to use
- pageSize - maximum number of users a thread will work with
- useDynamicPageSize - if set to true, (in this case pageSize is overwritten) each thread will work with an number of users equal to the total users divided by the number of threads.

It is the responsibility of the administrator to figure out the best values for the system. For instance if we have a quad core server dedicated to sipXecs with a few thousands of users, the default values should be fine. On the contrary if we have a dual core, then we'd want to leave the replication running on just a single thread. If we had a 8 core server with a lot of ram, allowing replication to work with 4 threads and setting *useDynamicPageSize* to true would probably give the best results.